ROD DSP software

ATLAS SCT and pixel off-detector electronics PDR

31 July, 2000

Damon Fasching

The University of Wisconsin, Madison

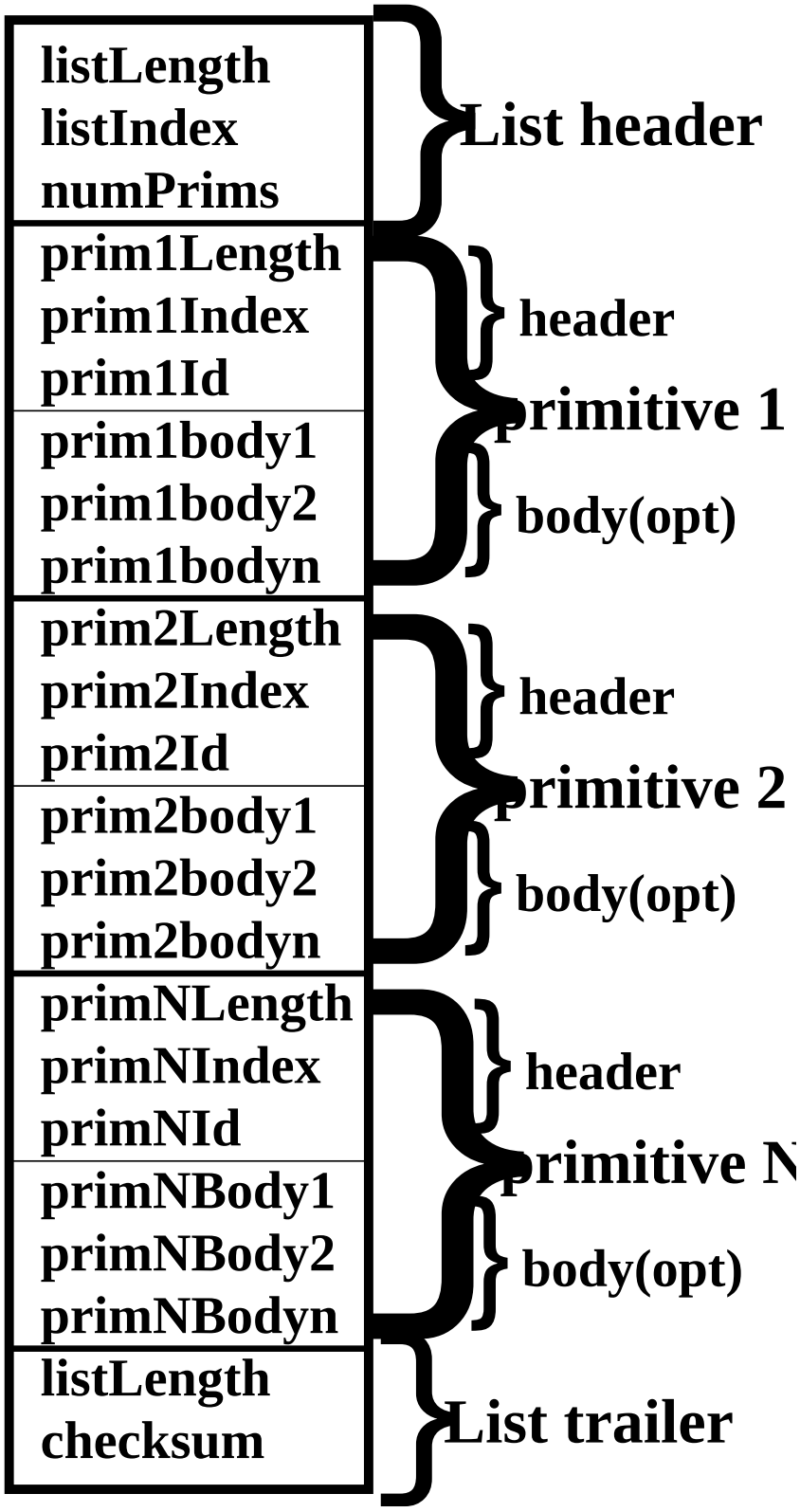
ROD crate software

- Much of this talk is distilled from the “protocol document.” (references at the end of presentation)
- Code is written according to “Suggested coding rules for SCT and Pixel ROD software” (J. Hill, based on “C++ Coding Standard Specification” adopted by ATLAS online).
- The ROD operating model is similar to that of the PLL. It will operate as a list processor, driven by lists of “primitive” commands downloaded from a VME host processor.
- An optimal set of primitives should
  - be a small number of commands which
  - allow a great deal of flexibility while
  - requiring few modifications or additions.
- Because the list processor on the ROD is a DSP, the communication protocols for exchange of information are more sophisticated.
- For each primitive, there is a corresponding function in the DSP code.
- The host has DMA access to the entire memory space of the DSP. Information is exchanged via buffers in the master DSP SDRAM (32 MByte). There are buffers set aside for primitive lists, reply data, error messages, etc.

ROD crate software

- There are two types of communication:
  1. transfer of primitive lists downstream and reply data upstream (data buffers)
  2. reporting error and information messages upstream (text buffers)
- Both protocols are implemented as state machines. Handshake bits in dedicated communication registers drive the associated state machine and also indicate its state.
  - command register: RW from VME, R from ROD
  - status register: RW from ROD, R from host
- At any time, a message may arrive from the host or one of the slave DSPs which initiates an action by the master. The masterDSP executes a loop which includes a call to each state machine. The state machine then executes an action depending on its state and the current conditions.
- Protocols at the host-masterDSP interface and masterDSP-slaveDSP interface are identical. Each slave also executes a polling loop and may receive messages from upstream which initiate actions.

$\Rightarrow$ Now a look at the code and protocols from the outside in.



Primitive list format

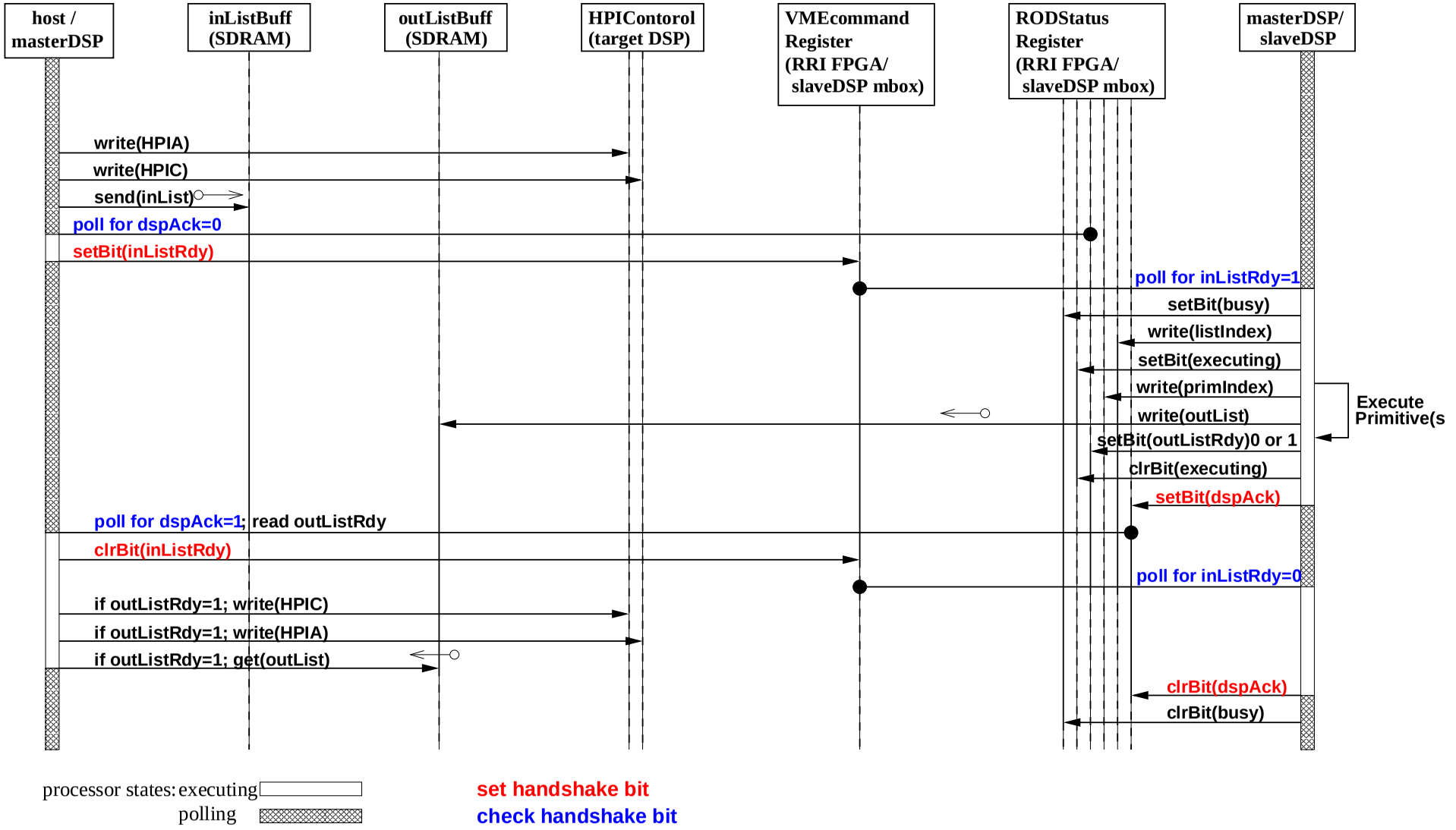
main routine in master DSP (so far only state machine calls)

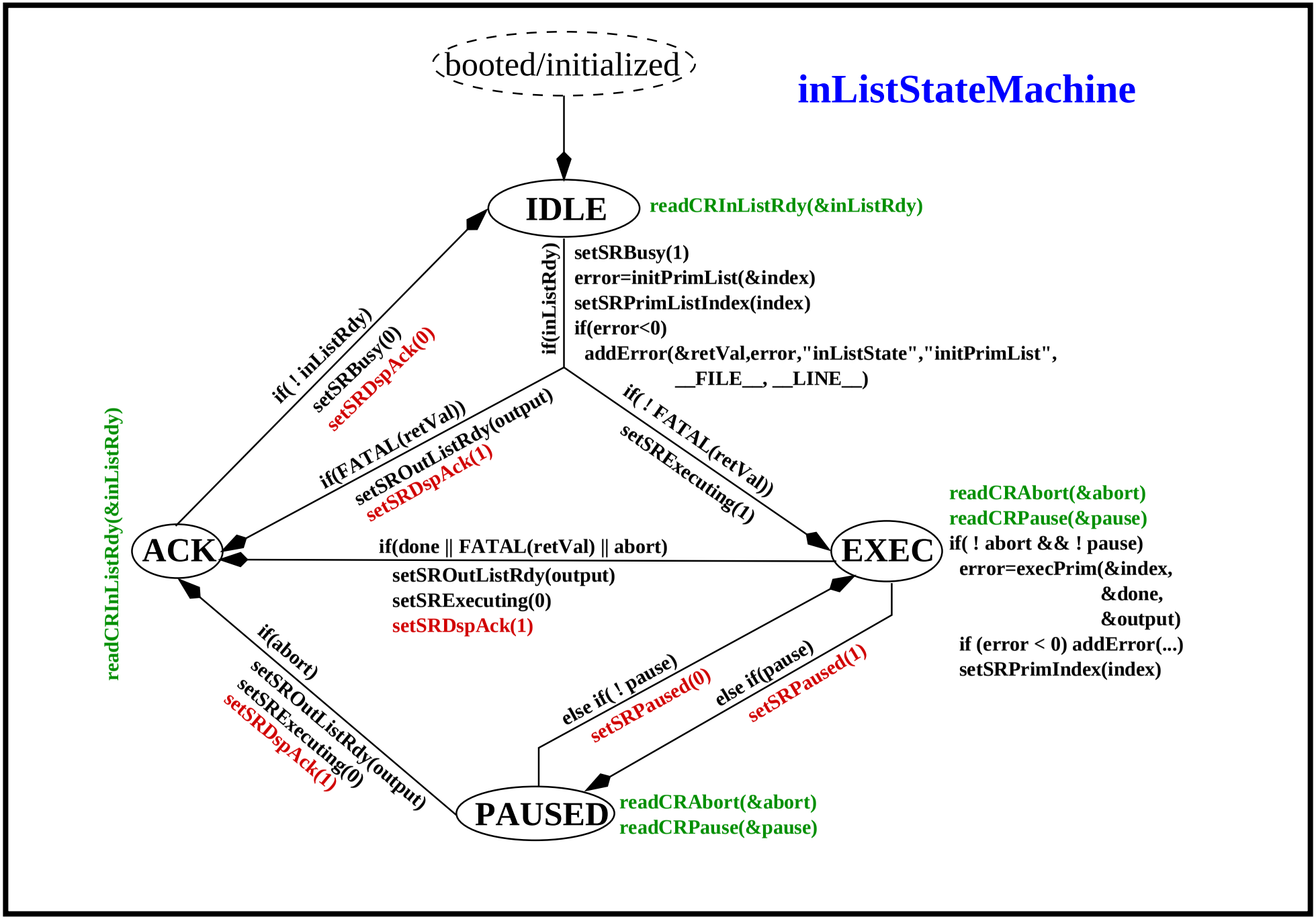
```
main() {
  SINT32 error, returnCode;
  UINT32 txtBuffLoop;

  error = initialize();
  if (error < 0) {
    addError(&returnCode, error, "main", "initialize", __FILE__, __LINE__);
  }
  setSRRunning(1);

  while(1) {
    error = inListState();                          /* primitive list state machine */
    if (error < 0) {
      addError(&returnCode, error, "main", "inListState", __FILE__, __LINE__);
    }
    for (txtBuffLoop = 0; txtBuffLoop < N_TXT_BUFFS; ++txtBuffLoop) {
      error = txtBuffStateMachine(txtBuffLoop);       /* text buffer state machine */
      if (error < 0) {
        addError(&returnCode, error, "main", "txtBuffState", __FILE__, __LINE__);
      }
    }
  }
  exit(SUCCESS);
}
```

Primitive list passing protocol; host to masterDSP and masterDSP to slaveDSP




booted/initialized
inListStateMachine
IDLE
readCRInListRdy(&inListRdy)
setSRBusy(1)
error=initPrimList(&index)
setSRPrimListIndex(index)
if(error<0)
addError(&retVal,error,"inListState","initPrimList",
__FILE__, __LINE__)
if(inListRdy)
if(! inListRdy)
setSRBusy(0)
setSRDspAck(0)
if(FATAL(retVal))
setSROutListRdy(output)
setSRDspAck(1)
if(! FATAL(retVal))
setSRExecuting(1)
readCRInListRdy(&inListRdy)
ACK
EXEC
if(done || FATAL(retVal) || abort)
setSROutListRdy(output)
setSRExecuting(0)
setSRDspAck(1)
readCRAbort(&abort)
readCRPause(&pause)
if(! abort && ! pause)
error=execPrim(&index,
&done,
&output)
if (error < 0) addError(...)
setSRPrimIndex(index)
if(abort)
setSROutListRdy(output)
setSRExecuting(0)
setSRDspAck(1)
else if(! pause)
setSRPaused(0)
else if(pause)
setSRPaused(1)
PAUSED
readCRAbort(&abort)
readCRPause(&pause)

primitive execution

- Now some implementation details of the primitive list handler.
  1. primParams.h
     - Synopsis: Defines prototypes of primitive functions, structure tags for primitive data and reply data, and primitive ids which are used to index the array of pointers to primitive functions.
  2. primFuncts.c
     - Synopsis: Contains primitive functions and the initialization routine for the array to pointers of these functions.
  3. inList.c
     - Synopsis: Contains functions which manage incoming primitive lists and outgoing reply lists, including data integrity and consistency checks. The routines in this file are one level below the primitive list state machine.
- We will look at examples from the above files which illustrate the structure of the primitive handler.

primitive execution

- primParams.h (header file on host and DSP)

```
/* primitive ids and structure tags */

#if defined(I_AM_ROD_CONTROLLER_DSP) || defined(I_AM_HOST)

#define ECHO              0
#define MEMORY_TEST       (1 + (ECHO))
      struct MEM_IN {
        UINT8 *startAddress;
        UINT8 *endAddress;
        UINT32 numReps;
      };
      struct MEM_OUT {
        UINT32 numErrors;
      };
#define ANOTHER_PRIMITIVE (1 + (MEMORY_TEST))
#define NUM_PRIMITIVES    (1 + (ANOTHER_PRIMITIVE))

/* primitive function prototypes */

#if !defined(I_AM_HOST)

int echo(struct PRIM_DATA *);
int memoryTest(struct PRIM_DATA *);
int anotherPrimitive(struct PRIM_DATA *);

#endif
#endif
```

- primFuncts.c (primitive functions)

```
void assignFunctPtrs(int (**primFunction)(struct PRIM_DATA *)) {

  primFunction[ECHO]              = echo;
  primFunction[MEMORY_TEST]       = memoryTest;
  primFunction[ANOTHER_PRIMITIVE] = anotherPrimitive;

  return;
}
```

- inList.c (executes primitives)

```
int (*primFunction[NUM_PRIMITIVES])(struct PRIM_DATA *);

int execPrim(UINT32 *index, UINT32 *listDone, UINT32 *outputData) {
  . . .
  copyMsgHead(&primHead, (struct MSG_HEAD *)primList.rwPtr);
  . . .
  *index = primHead.index;
  primData.priBodyLength = primHead.length - (SIZEOF(struct MSG_HEAD));
  primData.priBodyPtr    = primList.rwPtr + (SIZEOF(struct MSG_HEAD));
  primData.repBodyLength = 0;
  primData.repBodyPtr    = repList.rwPtr  + (SIZEOF(struct MSG_HEAD));
  primData.repBuffEnd    = (void *)((REP_BUFF_BASE) + (REP_BUFF_SIZE));
  . . .
    error = primFunction[primHead.id](&primData);
  . . .
```

- primFuncts.c

```
int echo(struct PRIM_DATA *primData) {
  UINT32 counter;
  for (counter = 0; counter < primData->priBodyLength; ++counter) {
    *(primData->repBodyPtr + counter) =
      *(primData->priBodyPtr + counter);
  }
  primData->repBodyLength = primData->priBodyLength;
  return SUCCESS;
}

int memoryTest(struct PRIM_DATA *primData) {
  SINT32 returnCode;
  UINT32 counter;
  UINT8  value *addr;
  struct MEM_IN  *memIn  = (struct MEM_IN *)primData->priBodyPtr;
  struct MEM_OUT *memOut = (struct MEM_OUT *)primData->repBodyPtr;
  returnCode = SUCCESS;
  srand(clock());
  value = rand() & 0xFF;
  memOut->numErrors = 0;

  for (addr = memIn->startAddress; addr <= memIn->endAddress; ++addr) {
    for (counter = 0; counter < memIn->numReps; ++counter) {
      *addr = value+1;
      if (*addr != value) ++memOut->numErrors;
    }
  }
  primData->repBodyLength = 1;
  return returnCode;
}
```

primitive execution

- Adding a primitive requires the following code.

1. primParams.h

   Define mnemonic and (optionally) structures for the host and the DSP.

```
#define YET_ANOTHER_PRIMITIVE (1 + (ANOTHER_PRIMITIVE))
      struct YAP_IN { . . . };      /* optional */
      struct YAP_OUT { . . . };     /* optional */
#define NUM_PRIMITIVES    (1 + (YET_ANOTHER_PRIMITIVE))
```

   Add the function prototype.

```
int yetAnotherPrimitive(struct PRIM_DATA *);
```

2. primFuncts.c

   In assignFunctPtrs, add the line

```
primFunction[YET_ANOTHER_PRIMITIVE] = yetAnotherPrimitive;
```

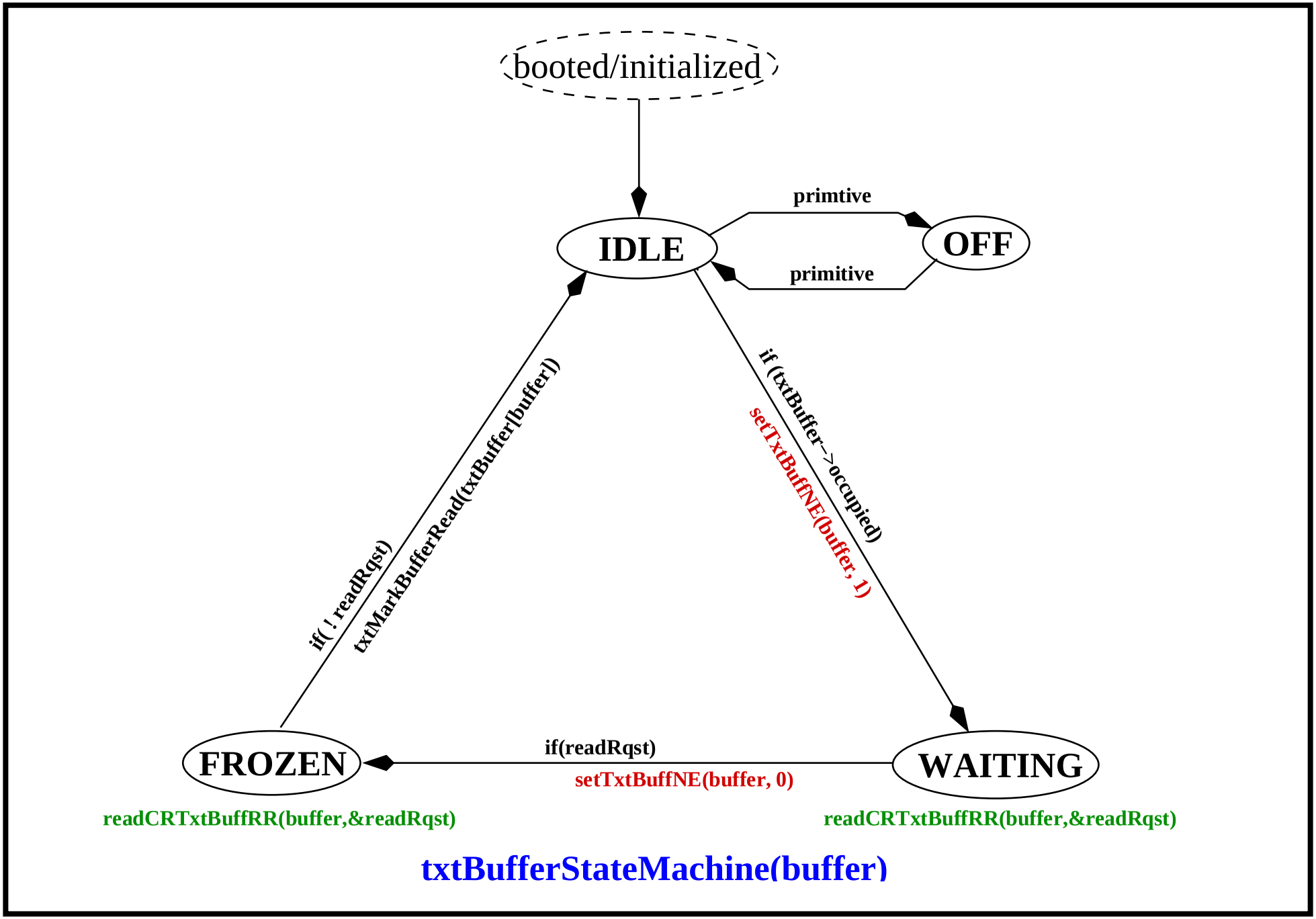
   Then write the routine(s).

```
int yetAnotherPrimitive(struct PRIM_DATA *primData) {
  return anInteger;
}
```

Error handling

- If an error condition is detected during the execution of a function about which a human user should be informed
  - An ASCII formatted message describing the error, including the file name, the line number and the function name is written to the error buffer.
  - If the error is sufficiently severe the function is aborted. Control is returned to the calling function with a fatal error code.
  - If the error is not fatal function execution continues. When completed, control is returned to the calling function with a non-fatal error code.
- If function receives an error return from a lower level function
  - An ASCII formatted message with the error code, the file name, the line number and the function names is appended to the error buffer.
  - If the error is sufficiently severe the function is aborted. Control is returned to the calling function with a fatal error code.
  - Otherwise, function execution continues. When completed, control is returned to the calling function with a non-fatal error code.
- In this way a complete calling stack and description of the error is built up in the error buffer.

- The next time the error buffer state machine is called, this information will be passed to the host processor. The information may be logged or sent to an online display if severity dictates.
- Two small service routines provide a uniform interface between all functions and the error buffer. This should facilitate error parsing by the host processor.
- This is not the final word on error handling. For example an error which is “fatal” from the point of view of one function may not be fatal from the point of view of the function which called it. A “retry” level has been suggested.


booted/initialized
IDLE
primtive
primitive
OFF
if (txtBuffer->occupied)
setTxtBuffNE(buffer, 1)
if(! readRqst)
txtMarkBufferRead(txtBuffer[buffer])
FROZEN
if(readRqst)
setTxtBuffNE(buffer, 0)
WAITING
readCRTxtBuffRR(buffer,&readRqst)
readCRTxtBuffRR(buffer,&readRqst)
txtBufferStateMachine(buffer)

DSP environments: ROD vs. Development

1. ROD
   - Communication bits are registers in the ROD resources interface FPGA.
   - Message buffers reside in SDRAM at CE2 and CE3 of DSP external memory interface.
   - Host writes prim list, reads reply and text buffers, sets bits in the VME command register.
   - DSP reads prim list, writes reply and text buffers, sets bits in the DSP status register.
2. TI evaluation module under host control (NI Lab Windows)
   - Communication bits are mailboxes in SDRAM, requires a single address change.
   - Message buffers reside in SDRAM at CE2 and CE3 of DSP external memory interface.
   - Host writes prim list, reads reply and text buffers, sets bits in the VME command “register.”
   - DSP reads prim list, writes reply and text buffers, sets bits in the DSP status “register”.
3. TI evaluation module stand-alone (TI Code Composer)
   - Communication bits are mailboxes in SDRAM.
   - Message buffers reside in SDRAM at CE2 and CE3 of DSP external memory interface.
   - DSP writes prim list, reads reply and text buffers, sets bits in the VME command register; i.e. the host is simulated.
   - DSP reads prim list, writes reply and text buffers, sets bits in the DSP status register.

$\Rightarrow$ The latter cases are each turned on via a single preprocessor directive.

$\Rightarrow$ Only a small portion of the very highest level DSP code is altered.

A sample of primitives from the protocol document

1. MasterDsp primitives

(a) Set CAL and L1A sequence parameters

This will be the method for setting parameters of the L1A and CAL commands for straightforward calibration and noise runs. A sequence is either CAL-L1A, CAL-L1A-L1A, L1A, or L1A-L1A. During the execution of these sequences, initiated by the ”Issue CAL and L1A sequences” primitive below, the interval between the sequences and the interval between the individual commands of a sequence are fixed by the following attributes.

Attributes:

i. number of sequences
ii. interval between sequences (End of sequence n to start of sequence n+1)
iii. interval between CAL pulse and 1st L1A. (0 = no CAL pulse)
iv. interval between 1st L1A and 2nd L1A (0 = no 2nd L1A)
v. event type (to trap data in proper DSP)

(b) Issue CAL and L1A sequences

This primitive causes the number and type of sequences described by the last ”Set CAL and L1A sequence parameters” to be issued. It is assumed that the CAL and L1A sequence parameters, the data path, FE_CMND_MASK,

the SlaveDSPs, and the detector mounted electronics have been set up prior to issuing this primitive.

2. SlaveDsp primitives

(a) Setup for calibration

Attributes:

i. fitting function
   A. 1: no fit, keep raw histograms
   B. 2: S curve
   C. 3-9: reserved
   D. 10+n: nth order polynomial

ii. X axis source (for pixels address mapping needs to be worked out)
   A. element number
   B. element number + TOT (pixels only)
   C. element number + control variable (e.g. DAC step)

iii. Y axis maximum
   A. 1: 1 byte (256 counts)
   B. 2: 2 bytes (65K counts)

(b) **Setup for error counter and event synchronization correction**

Attributes:

i. resynchronization mode
   A. 1: off (do not perform resynchronization)
   B. 2: on
ii. alarm threshold (may want to define by error type)

(c) **Setup for event trapping (monitoring events)**

(d) **Setup for occupancy plots**

Attributes:

i. Y axis maximum
   A. 1: 1 byte (256 counts)
   B. 2: 2 bytes (65K counts)
ii. number of events (-1 = accumulate same number as in the reference occupancy histograms)
iii. alarm tolerance
   A. -1: send raw hists (do not compare with reference)
   B. s: alarm at s sigma deviation from reference

(e) **Accumulate reference occupancy histogram**

Attributes:

i. Y axis maximum
   A. 1: 1 byte (256 counts)
   B. 2: 2 bytes (65K counts)
ii. number of events
iii. date

(f) **Read parameters (Y axis maximum, number of events, date) of current reference occupancy histograms**

(g) **Process calibration histograms**

Attributes:

i. fitting function
   A. 2: S curve
   B. 3-9: reserved
   C. 10+n: nth order polynomial
ii. scan parameters (This should include things like the number of pulses per point, the axis values of the points, etc. From the host side, it may be easier if these parameters are sent down when the scan is being set up rather than when it is finished. People involved in that end of things should comment.)

(h) **Clear histogram(s)**

Attributes:

i. histogram number (-1 to clear all)

Software Status

- host-masterDSP and masterDSP-slaveDSP communication protocol designed
- communication code written (state machines and buffer management)and tested between host and masterDSP
  - DSP side tested stand-alone using an evaluation module, EVM, from Texas Instruments with the host side simulated. The EVM is a PCI bus card available from TI. It is designed around the target DSP and facilitates code development, benchmarking, etc.
  - host side tested with up to 20 simulated DSPs (L. Tomasek's presentation)
  - host and DSP side tested together with a single DSP EVM (L. Tomasek's presentation)
- standard error handling procedure designed and implemented
- standard procedure for including primitive functions designed and implemented
- list of primitive functions has been written down and iterated

Next Steps

- implementation of DSP functionality ongoing
  - runtime and calibration: coordination of tasks in master DSP
  - error diagnostics, histogram and fit routines in slave DSPs
  - study flow of data which is “off the main path”, e.g. monitoring data @ 1 kHz.
  - fleshing out and implementation of primitive functions
  - slave DSP code development and benchmarking
- test stand development through end of August
- fabrication and loading of boards through mid September
- stand-alone ROD tests through $\approx$ end September (in parallel with loading)
- off-detector system test during November in the UK (one month of cushion)
- user evaluation in the following months, depending on availability of modules
- provide pixel and SCT local DAQ (ROD crate DAQ)

some web references

- http://www-wisconsin.cern.ch/~atlas/off-detector/off-detector.html
  - meeting minutes
- http://www-wisconsin.cern.ch/~atsiod/shared.html
  - coding rules document
  - message protocol document, includes current iteration of primitives